Real Time Control of a Magnetic Bearing Suspension System for Flexible Rotors

Edgar F. Hilton, Research Associate*
Marty Humphrey, Research Assistant Professor†
John Stankovic, B.P. Professor and Chair‡
Paul Allaire, MacWade Professor§

University of Virginia
Charlottesville, VA 22901

November 30, 1999

Abstract

Suspension of a magnetic bearing system involves significant computational effort. Control systems based upon embedded Digital Signal Processors (DSP) boards often require specialized programming and development tools, may lack flexibility when computational requirements change, and are often relatively expensive. Magnetic bearing systems need 1) real time monitoring of the plant states, inputs, and outputs 2) real time plotting functions such as rotor position *vs.* time, Fast Fourier Transform (FFT) functions and user specified filters, 3) controller parameter updates, and 4) access to reference signals. It is desired to employ a hardware/software system which is low cost, easy to use, is extensible as more advanced versions of hardware/software become available, and distributable over local networks of DSPs. A system with all of the above desired characteristics has been implemented for control of a magnetic bearing supported flexible rotor using RT-Linux, a free modification of Linux, intended to support hard real-time computation. Experiences designing the software architecture, defining timing requirements of the control tasks, implementing the control tasks in RT-Linux, and measuring the predictability of RT-Linux for this application are discussed.

1 Introduction

An artificial barrier seems to exist that separates the implementation of complex automatic control systems on active magnetic bearings (AMBs) and basic scheduling theory as devised in computer

*efhilton@alum.mit.edu, Mechanical and Aerospace Engineering, Rotating Machinery and Controls Laboratory, UVA
†Department of Computer Science, UVA
‡Department of Computer Science, UVA
§Mechanical and Aerospace Engineering, Rotating Machinery and Controls Laboratory, UVA

science circles. This paper seeks to remove that barrier and discuss the priority driven scheduling algorithms that are needed to efficiently implement complex automatic controllers for AMBs.

AMBs now require multiple computational tasks such as (in order of importance) 1) periodic fixed rate closed loop suspension loops, 2) a spin rate measuring system, 3) open loop balancing controller, 4) data transfer and plotting tasks, 5) network transfer tasks, and 6) miscellaneous additional tasks such as screen refresh or shell programs. Commonly, each of these tasks is implemented digitally as a sequence of commands which are interpreted by a digital computer, one command at a time. Normally, all of these tasks are easily implemented if enough independent computational engines are available unless restricted by factors such as hard disk, network, or bus access times (Stankovic, 1988; Stankovic and Buttazzo, 1995).

However, many applications of AMBs have a need for highly efficient inter-task communications, controller weight limitations, controller size limitations, cost limitations, or other factors (Allaire et al., 1994). Thus, the solution is to implement all of these tasks in one single CPU by the use of Real Time systems, using some of the many optimal scheduling algorithms that are currently available in this field.

Full embedded control for an AMB system usually requires significant computational effort – especially for flexible rotors. This is caused by the inherent instability of all AMBs. That is, all AMBs are open loop unstable. Thus, the only way to stabilize an AMB system is via a properly designed closed loop controller. However, in order to correctly tune an AMB controller, the controls engineer needs to fully evaluate AMB performance via considerable access to plant input/output (I/O), controller states, and controller parameters. Most importantly, and for safety reasons, in high speed AMB applications the controls engineer needs to get access to this data from a safe location which may or may not necessarily be even in the same building.

The success of the AMB is heavily dependent on the design of the controller. In turn, the controller relies heavily on *a priori* knowledge of the plant dynamics. Thus, considerable modeling, characterization, and controller parameter calibration effort is necessary during the early controller implementation stages for a given AMB application. However, once the controller has been selected and tuned for the particular bearing, the controller is most often not revised.

An important aspect of real time computing for magnetic bearings is the effectiveness of resource allocation strategies so as to satisfy stringent timing-behavior requirements (Stankovic, 1988). The proper design of a real time control system requires solutions to many interesting problems - for example, specification and timing behavior, and programming languages semantics dealing with time, and the use of timing constraints. The correct functioning of the system depends upon an implementation which evaluates the logical power of different forms of timing constraints in solving various coordination problems and determines the least restrictive timing constraints sufficient for the control system. Unlike other combinatorial scheduling problems in operations research which mostly deal with one shot tasks, in real time control systems, the same task may recur very often, either periodically or at irregular intervals, and thus may have to synchronize or communicate with a number of other tasks (Stankovic and Buttazzo, 1995) This is the case with AMB systems such as magnetic bearing supported artificial hearts (Hilton et al., 1999; Baloh et al., 1999; Allaire et al., 1996) and high speed energy storage flywheels for powering communication satellites (Bartlett et al., 1999).

The primary objectives of real time systems design for automatic controls include 1) automation of the process by exploiting optimizing transforms and scheduling theory and 2) the synthesis of highly efficient code and customized resource schedulers from timing constraint specifications. Reliance on clever hand coding and difficult to trace timing assumptions are major sources of bugs in real-time

programming that can be avoided with recent advances in real time structured computing resources such as Linux and Real Time Linux. Real Time Linux is an add-on to the Linux operating system which converts the Linux OS into a hard real time environment by implementing any of many powerful Real Time scheduling algorithms.

The central idea in real time systems is to allocate resources judiciously to make certain that any critical timing constraints can be met with the available resources, assuming that the hardware/software functions correctly and the external environment does not stress the system beyond what it is designed to handle. This is best handled via powerful scheduling algorithms. The real time systems literature is filled with different types of scheduling algorithms, both static[1] and dynamic[2]. This paper does not attempt to review all of these algorithms since some of these do not lend themselves too well for AMB applications. Instead, this paper concentrates on one of the so called "static" scheduling algorithms, the "Fixed Priority Scheduling". This scheduling algorithm is especially useful for periodic tasks, as would be the case in the implementation of most fixed rate controller algorithms.

The following sections are broken up as follows. First, Real Time Systems and their associated basic scheduling results are presented. Second, the Real Time Controls Laboratory – a controls implementation platform developed and tested at the University of Virginia Rotating Machinery and Controls Laboratory – is presented along with the underlying Real Time Linux executive. Finally, suspension results are shown while using the Real Time Controls Laboratory.

2 Real Time Systems

"Real Time Systems", are computer systems in which the temporal correctness of the system is at least as important as the logical correctness of those results. For example, in a high speed AMB, it is imperative that our fixed rate suspension controller maintains a strict sampling rate of 5 kHz, irrespective of whether or not the screen and graphical user interface have been fully refreshed. Failure to do so could cause our AMB to catastrophically fall out of suspension at the mere touch of the computer mouse.

Unfortunately, there are several misconceptions (Stankovic, 1988) regarding Real Time Systems which have inhibited the controls community from correctly implementing real time controllers in AMBs. Four of the perhaps most commonly cited misconceptions in our field are:

1. *faster hardware implies that all deadlines will be met*: while it may be true that faster hardware will minimize the mean response time of our system, it does not necessarily imply that the system will be predictable (Stankovic and Ramamritham, 1990), that is, that it will execute precisely at the requested sample rate.

2. *real time systems are equivalent to control systems programmed in assembly coding, interrupt programming, and complex device drivers*: one of the most important research aspects of real time systems is that researchers concentrate on developing powerful scheduling algorithms that will meet all hard timing constraints. Consequently, a real time system designer can now use high level code such as Ada and C instead of a more arcane and platform specific assembly language.

[1] The scheduler has full knowledge of previous, present, and future tasks, as would be the case for a fixed rate controller.

[2] The scheduler has full knowledge of previous and current tasks, but does not know the time nor the number of tasks that will arrive in the future.

3. *real time systems are all developed in an* ad hoc *fashion*: real time systems research concentrates on developing powerful, flexible, and structured techniques that formalize the actual development and implementation of real time systems. Many structured tools now exist to help develop, validate, and simulate real time systems.

4. *real time is equivalent to fast computing*: "fast" is relative. That is, in the AMB community, a sampling rate of 100 μs is considered "fast". In the robotics community, a sampling rate of 1,000 μs is already considered "fast". In the geo-sciences community, a sampling rate of 86,400,000 μs (1 day) is considered "fast". In all three systems, it is imperative that tasks execute at *precisely* the given time or else the results may no longer be valid. Consequently, all three systems are categorized as "real time systems".

The real time systems community focuses on many aspects of real time research. Namely, they consider real time hardware, software, validation, and simulation among others. We, as end users, normally do not attempt to understand all of these and rely on the Computer Science community to develop most of these technologies. However, it is imperative that we understand, as a minimum, some very powerful scheduling results which greatly aid in the design process of real time systems and consequently in the implementation of hard-timing control environments. These results are discussed in what follows.

2.1 Scheduling Theory

One of the first works on scheduling theory for computer systems appeared in 1967 by Conway, Maxwell, and Miller (Conway et al., 1967). In this work, they reference much scheduling work that up to that point had been primarily used in job shops across the country. They discuss results which are applicable to both what later became known as classical scheduling theory, and real time systems.

Classical scheduling theory – such as what would normally be seen in a regular multitasking operating system such as Unix – typically uses metrics such as minimizing the sum of completion times, minimizing the weighted sum of completion times, minimizing schedule length, or minimizing the number of processors required. Most of these metrics evolved in job shops during and immediately after World War II, where the main concern was of maximizing the throughput of a given machine, such as either a lathe or a milling machine.

Coffman and Denning (1973) showed that these metrics are not useful for Real Time Systems. For example, the sum of completion times is useless for time critical systems because there is no direct assessment of timing properties (deadlines or periods) in the metric. Thus this algorithm may be desirable for Unix, where the goal is to maximize the throughput or number of tasks that are executed in a given amount of time, but will not be desirable for a hard timing environment as would be the case of AMB control.

Real time researchers thus looked at the second results presented by Conway *et al.*. Namely, they cite a 1955 job-shop scheduling result by Jackson, where he states:

> *The maximum lateness and maximum job tardiness are minimized by sequencing the jobs in order of non-decreasing due dates*

This algorithm, which is usually referred to as the "Earliest Deadline First" or simply EDF algorithm, has been shown to be optimal for uniprocessor systems, where optimality in real time scheduling algorithms is measured by the following:

An optimal scheduling algorithm is one that may fail to meet a deadline only if no other scheduling algorithm can meet it.

The Jackson result sparked much interest and consequently spawned two areas of real time scheduling research. Namely, it developed research in both *dynamic* and *static* scheduling, the latter of which is more applicable to AMB control. A scheduling algorithm is said to be "dynamic" when the algorithm has full knowledge of task properties[3] of both previous and present tasks. However, it has no knowledge of future tasks, neither in the number of tasks nor in their start times.

A static scheduler, on the other hand, knows the task properties of all past, present, and future tasks and therefore it is possible to set up some scheduling parameters *a priori*. For example, suppose that an AMB will use a simple PID for thrust control and a plotting package for real time monitoring of plant states. We know *a priori* that the suspension controller must execute every 100 μs without fail, and that the plotting package will execute once every 14,000 μs, although we are not too concerned if one time it executes after 14k or after 50k μs. Consequently, we can assign, *a priori* some schedule parameters such as the next execution time (now + 100 μs and now + 14k μs), and a relative priority (it is more important that the suspension controller executes at its specified time).

A "fixed priority scheduler" (FPS) is a type of static scheduling algorithm where tasks are assigned – *a priori* – a priority, or relative importance, as shown in Figure 1. Each task has an associated start time at which the task is expected to begin its execution, an execution time, and a completion time. Thus, at both the start time and completion times of each of these tasks, the scheduler must make a decision. That is, it must decide which task should be allowed to execute in the CPU. Higher priority tasks are given precedence over lower priority tasks, and tasks having the same priority level are assigned into the CPU on a First In First Out (FIFO) scheme. Tasks may *preempt* lower priority tasks, only, and tasks in general are assumed to be completely independent of each other. Research in FPS algorithms is most heavy in optimal solutions to the priority assignment for each of these tasks, since depending on how priorities are assigned will determine if deadlines are met.

Liu and Layland (1973) developed the "Rate Monotonic Algorithm" (RMA). This algorithm assigns priorities to each of the tasks in the following fashion:

The priority of the task is inversely proportional to its period.

In other words, under this assignment policy, tasks having the smallest period (highest frequency) – as would be the case for a suspension controller in an AMB – would have the higher priority over a second task that executes with a larger period. Of most importance is that this algorithm was shown to be *optimal* among FPS algorithms for real time systems, and that it can be applied for any number n of tasks.

Unfortunately, the RMA algorithm was shown to be optimal only for periodic and *independent* tasks. However, since there is a large need for communication between controller tasks in an AMB (for example, controller effort from the suspension task are communicated to a second task which in turn transmits this information over the network), the RMA algorithm can be shown to be limiting for real time applications. Consequently, research ensued which tried to relax this independence assumption, necessary for RMA.

[3]example: execution time, start time, number of tasks

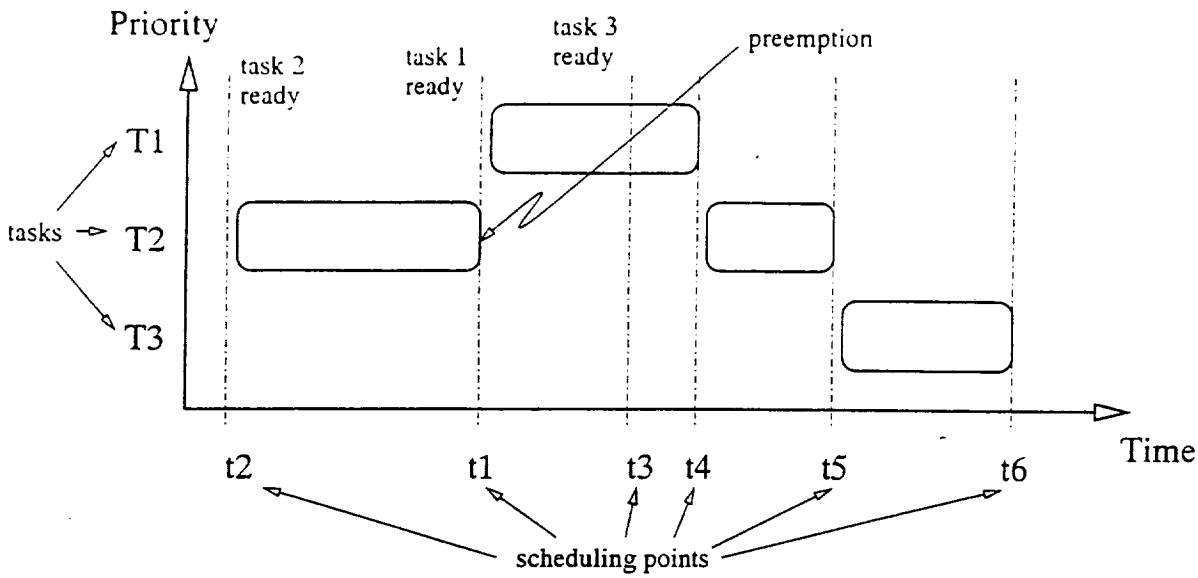

Figure 1: Fixed priority scheduling algorithm example. For illustration purposes, let T_1 denote a suspension controller, T_2 an anti-imbalance controller, and T_3 a plotting package.

The direct application of a simple *semaphore*[4], or synchronization variable, for sharing critical data between tasks may result in a high priority task being blocked by a lower priority task for an unbounded amount of time. More importantly, this may potentially lead to missed deadlines. This type of unbounded blocking is usually referred to as *priority inversion*, and is exemplified in Figure 2.

Priority inversion usually occurs when there are three or more tasks present, and data is shared between both the highest and lowest priority tasks. Generally, the lowest priority task may get hold of the semaphore and begin accessing data. Then, sometime later, the highest priority task will attempt to access the semaphore but will find that it is already locked by the lowest priority task, and therefore the highest priority task is blocked by the lowest priority task. At this time the lowest priority task resumes execution. However, some time later, the medium priority task begins executing and preempts the lowest priority task. Consequently, the highest priority task not only needs to wait for the medium priority task to finish, but also for the lowest priority task to release the semaphore. The problem becomes more acute when there are more than one medium priority tasks. Consequently, the highest priority task will have to wait for an undetermined amount of time until all intermediate priority tasks complete, and the lowest priority task releases its semaphore.

A powerful solution to this problem was introduced by Sha et al. (1990) under the name of "priority inheritance protocols" (PIPs). PIPs are a series of protocols in which the lower priority tasks that may be blocking a higher priority task will, while blocking the higher priority task, automatically inherit the priority of the highest priority task that the offending task blocks. Two protocols were presented: a basic priority inheritance protocol and a priority ceiling protocol. In both cases, priority inversion

[4]A semaphore is a "lock" which is shared between several tasks. Semaphores are primarily useful in protecting shared data – or critical sections of code –, where a requesting task may first lock the semaphore, access the data, and then release the semaphore. Only one task at a time may enable a semaphore. If any other tasks try to enable the semaphore, they will be put to sleep until the task currently owning the semaphore releases the semaphore, at which time one waiting task will be able to lock the semaphore and all other waiting tasks will continue to sleep.

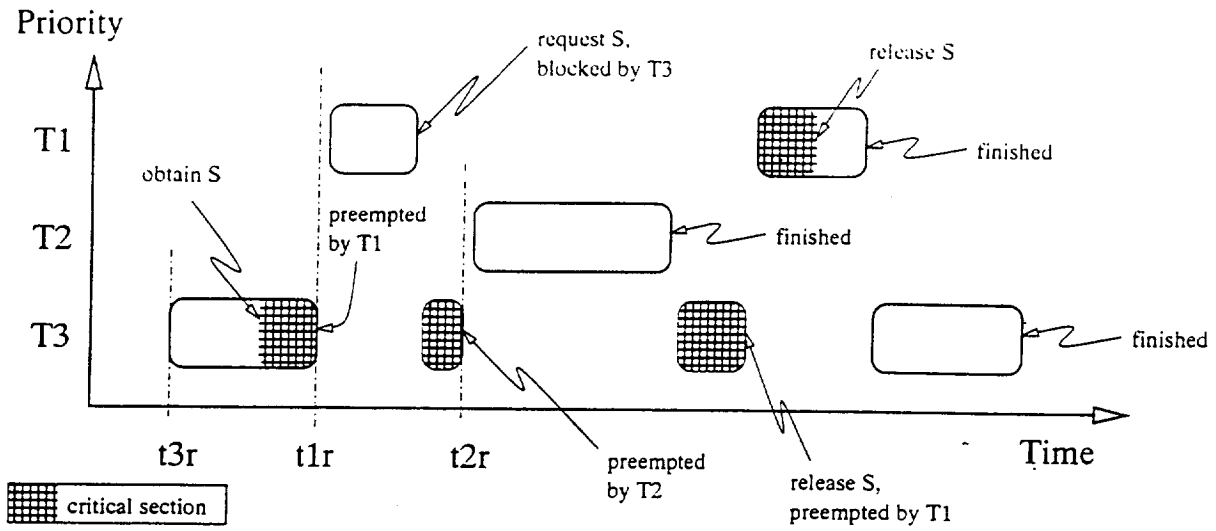

Figure 2: Fixed priority scheduling algorithm example. For illustration purposes, let T_1 denote a suspension controller running at a period of 100 μs, T_2 an anti-imbalance control algorithm, and T_3 a plotting algorithm. Tasks T_1 and T_3 share data, of which T_1 produces the data and T_3 plots it to screen. T_1 and T_3 share semaphore S which is used to protect the integrity of the data.

will be solved by effectively *bounding* the maximum amount of time that the highest priority tasks will be blocked by the lower priority tasks. The former will block the higher priority task by *at most* the duration of one critical section from *each* of the lower tasks which share semaphores with the higher priority tasks, while the latter will block the higher priority task by *at most* the duration of *one* shared segment from any of the lower priority tasks. Thus, by the application of either of these protocols, one can know, *a priori* the worst case execution time of any tasks.

2.2 Schedulability Analysis

Real time scheduling of periodic tasks (such as AMB controllers) often uses the rate-monotonic algorithm (Liu and Layland, 1973; Stankovic and Buttazzo, 1995). This algorithm has been shown to be optimal in the literature for uniprocessor systems and periodic tasks. The most important feature about this algorithm is that it introduces the ability to find, *a priori*, whether or not a given set of tasks will be schedulable – that is, whether or not they will meet all their deadlines.

Liu and Layland showed that for n independent tasks – all of which have the same start time at some point in time, whose execution time is smaller than their deadline, and which, most importantly, are completely independent one of the other – all tasks are schedulable if the following is true:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1) \tag{1}$$

where C_i is the worst case execution time of the task, P_i is the period of the task, and the left hand side of (1) is called the total CPU utilization. Of most use is the fact that:

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 69.34\% \tag{2}$$

or equivalently, if CPU utilization is less than 69.3%, all tasks will meet their deadlines. This relation describes a worst case sufficient condition for a rate-monotonic algorithm and is thus pessimistic. An exact characterization was obtained by Lehoczky, Sha, and Ding (Lehoczky et al., 1989) but is not quoted here due to its higher degree of complexity.

Equation (1) provides conditions under which a set of periodic tasks can be scheduled taking into account the effect of a task being preempted exclusively by higher priority tasks. However, (1) can be extended to include blocking of higher priority tasks by lower priority tasks (Sha et al., 1990). That is, assuming that the priority ceiling protocol is used to eliminate priority inversion, then (1) can be extended as:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + max\left(\frac{B_1}{P_1}, \ldots, \frac{B_{(n-1)}}{P_{(n-1)}},\right) \leq n(2^{1/n} - 1) \tag{3}$$

where B_j denotes the worst case blocking time that may occur from any of the lower priority tasks (where increasing j denotes tasks with decreasing priority).

The most important point to note about these scheduling tests is that we now have an effective method of selecting a CPU for a given application (or determining that no CPU will handle the given controller at the given sampling rate). That is, if we know the target execution periods (for example, a controller may repeat itself every 100 μs) and the worst case execution time for each of our control tasks (for example, the execution time of the aforementioned controller may be 80 μs), then by use of the aforementioned schedulability tests, we can scientifically select a computer that is appropriate for our application. Stated differently, if the left hand side of these tests is much smaller than the right hand side, then we know that the target CPU is much too fast for our application and thus we can select a lower cost system. Alternatively, if the left hand side is too large compared with the right, then we should either consider upgrading to a faster CPU or redesigning the controller given that our given CPU cannot compute this controller at the given rate). From our example and (1), we find that $80/100 = 0.8 \leq 1.0$, therefore our individual task is schedulable in this CPU and will meet all of its deadlines.

Real Time Systems literature is too vast to be summarized in this paper. Interested readers are encouraged to pursue further reading in many of the excellent papers on the subject.

In what follows, an AMB control solution has been implemented at the University of Virginia Rotating Machinery and Controls Laboratory in a novel controls implementation platform using Real Time Linux and a set of networked personal computers.

3 The Real Time Controls Laboratory, (RTiC-Lab)

Control of AMBs may require an exhaustive tuning process during the early stages of the AMB life. During this stage, the controls engineer performs thorough plant characterization and evaluation. Using this new-found information, the controls engineer can then proceed to tune the controller. For that,

the controls engineer needs soft real time access to plant I/O, controller states, controller parameters, and set points.

The *Real Time Controls Laboratory, RTiC-Lab*, is explicitly designed by the main author of this work to be used during these early stages of controller design, plant characterization, subsequent monitoring, and subsequent control. Designed and tested at the University of Virginia's Rotating Machinery and Controls Laboratory, it provides an environment in which to implement controller algorithms while providing real time access to controller states, plant outputs, controller actions, controller parameters, and other controller information. All this information can be plotted and filtered – via user defined filters – in soft real time. The user can further filter the necessary data either in real time or *post mortem*. Last and most importantly, the controller parameters can be updated in real time through a user-defined graphical user interface.

RTiC-Lab attempts to incorporate several of the most critical scheduling methods in order to make RTiC-Lab a powerful controls implementation platform for AMBs and any other system that can use both fixed rate and event driven controllers. Priority assignment has been employed with Liu and Layland's RMA scheduling algorithm. Data is synchronized via priority inheritance protocols, and is transmitted from the hard real time tasks to the graphical user interface via real time FIFOs.

RTiC-Lab has two very important features not found in any other real time controls implementation platforms. The first and most important one is that RTiC-Lab is and will be – as its underlying Linux and Real Time Linux platforms – Open Source Software. That is, users of RTiC-Lab can download the source code via the web from `http://www.people.virginia.edu/~efh4v`, and are heavily encouraged to both add to, and improve this code[5]. The second feature is that RTiC-Lab is designed to be distributed over a common network of personal computers. That is, RTiC-Lab can be used over a common 10/100 Mb ethernet network.

In conclusion, RTiC-Lab has been designed to be free, extremely flexible, organic, and powerful enough to handle very large tasks. Its success is based on the hard timing capabilities of Real Time Linux and the Open Source design of Linux.

3.1 RTiC-Lab Design

The general scheme used in the design of the Real Time Controls Laboratory is shown is Figure 3. A devoted display or host computer (DHC) is networked via 10 or 100 Mb/s TCP/IP network to a set of devoted controls computers (DCCs).

The controls engineer sits at the DHC (which may or may not be at the same room or even building as the DCCs) and coordinates, codes, and synchronizes all DCCs from the DHC. Run time parameters, such as sampling rate, startup delay, and networking parameters can be set for each of the DCCs from the DHC.

Each of the DCCs is a stripped down computer system having no keyboard, mouse, video card, or monitor. These only have both the necessary I/O cards which are used to interface to the plant hardware and the necessary ethernet card to communicate with the DHC.

An AMB example of RTiC-Lab is shown in Figure 4. A single DHC interfaces with three DCCs which in turn interface to the AMB rotor system. The first DCC handles all radial control of the AMB, while a second (and slower) DCC controls the thrust direction of the AMB, and a third DCC is used to add either some excitation or synchronous forces to cancel out rotor imbalances at the midspan. Both

[5]Please send all code submissions to the main author of this paper.

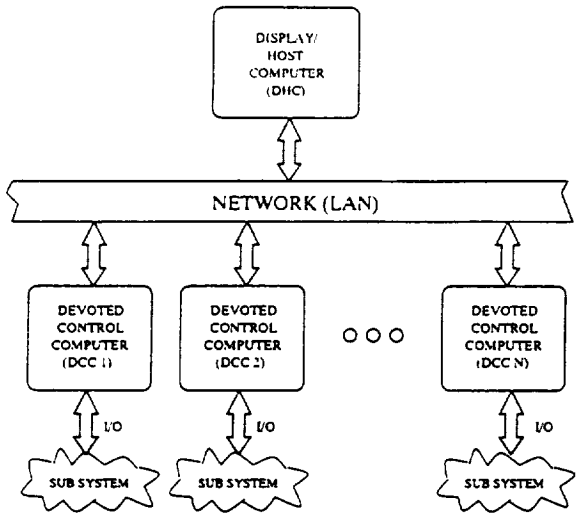

Figure 3: Overview of the Real Time Controls Laboratory network environment

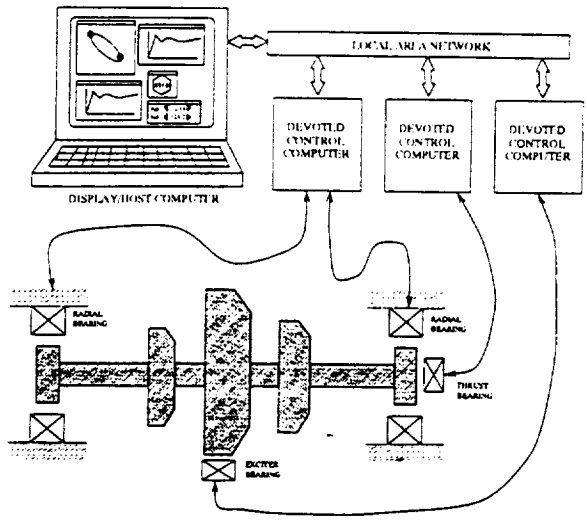

Figure 4: Example of the applicability of RTiC-Lab on an AMB system.

controller parameters can be updated through the graphical user interface, and all data is plotted in soft real time at the DHC.

In the event that the controlled plant is computationally simple enough to be handled exclusively in a single computer, then RTiC-Lab will collapse into one single computer to control the entire plant. Stated differently, the same computer is used to both plot all incoming data and to update controller parameters in soft real time.

In accordance with the RT-Linux paradigm, RTiC-Lab separates the AMB controller into the hard real time or "embedded" part and the soft real time or "reactive" part. The embedded part of the controller (resident exclusively in the DCCs) includes all tasks having hard timing constraints:

1. the AMB suspension controller(s) (both periodic and event driven),
2. a software watchdog, and
3. a set of interrupt service routines that are used for communication with the reactive task.

The reactive task (resident in both DHC and DCCs) is a multi-threaded, user-space application which runs within the Linux kernel. In a standalone system, the reactive task would perform the following functions:

1. communicate with the embedded tasks via RT-FIFOs,
2. display a graphical user interface for the user,
3. perform error checking of the user's controller code,
4. send parameter updates to the embedded tasks as requested by user,
5. plot data to either screen, save to a file, or print to stdout,

Alternatively, in a multi-node environment, the DCCs' reactive system is charged with communicating with both the local embedded tasks via RT-FIFOs and with the remote DHC through the LAN. It would also be used to trap some vital errors from the local real time tasks, such as missed deadlines. The DHC - which does not necessarily have real time support - would then receive the incoming data through the network and would perform all of the necessary graphical duties as described above. In addition, the DHC would be used to coordinate all networked DCCs.

3.2 RTiC-Lab Proof of Concept

Figure 5 shows the rotor for the UVA/ROMAC controls test rig (CTR). Some salient points about this rotor system that make this system an excellent test bench for the Real Time Controls Laboratory are itemized as follows:

- rotor weight is 37 lbf, consequently emphasis is placed on minimization of rotor vibration and force transmissivity into the underlying frame and housing.
- $D/L << 1$, that is, this rotor is long and skinny and thus has many flexible modes that must be controlled during normal rotor operation.
- the surrounding housing and casing are relatively flexible and further introduce dynamics of their own.
- due to the high frequency content in the sensor and actuator wiring harnessess, all cabling must be kept as short as possible in order to minimize electrical noise in the system. This implies that the computational engine must be placed as close to the rig as possible. Yet for safety reasons, the controls engineer should remotely monitor the rig from any location outside of the test chamber.

As of the time of this writing, a single *Pentium II, 450 MHz* computer was used to implement a full five degree of freedom controller sampling at 5 kHz. The thrust direction is controlled using a strictly proper PID controller, and each of the remaining radial control currents (X and Y for an upper and lower magnetic bearing) are controlled using strictly proper PID controllers alongside four notch filters per channel, as shown in Figure 6. For this application, a total of 61 controller parameters could be updated in soft real time through RTiC-Lab.

The RTiC-Lab windows were exported (using the XWindow ability to do so) from the control computer to a second Linux computer housed in a remote location, from which we could safely both monitor the rotor system in the test chamber and control the embedded tasks in the controls computer. Consequently, the test chamber was evacuated during the rotor spin tests. All sampled data was stored in the control computer's harddrive and downloaded via ftp to the remote computer in "pseudo real time".

3.3 Suspension Results

Figures 7 and 8 show the suspension results of the controls test rig at 5,000 RPM. In the figures, the rotor vibrates less than 0.2 thousandths of an inch from its nominal location[6]. The first bending mode is expected at 8,000 RPM and presently measures are being taken to take the rotor up to 12,000 RPM before the end of the year.

[6]Note that as of the time of this writing, the rotor has not been formally balanced

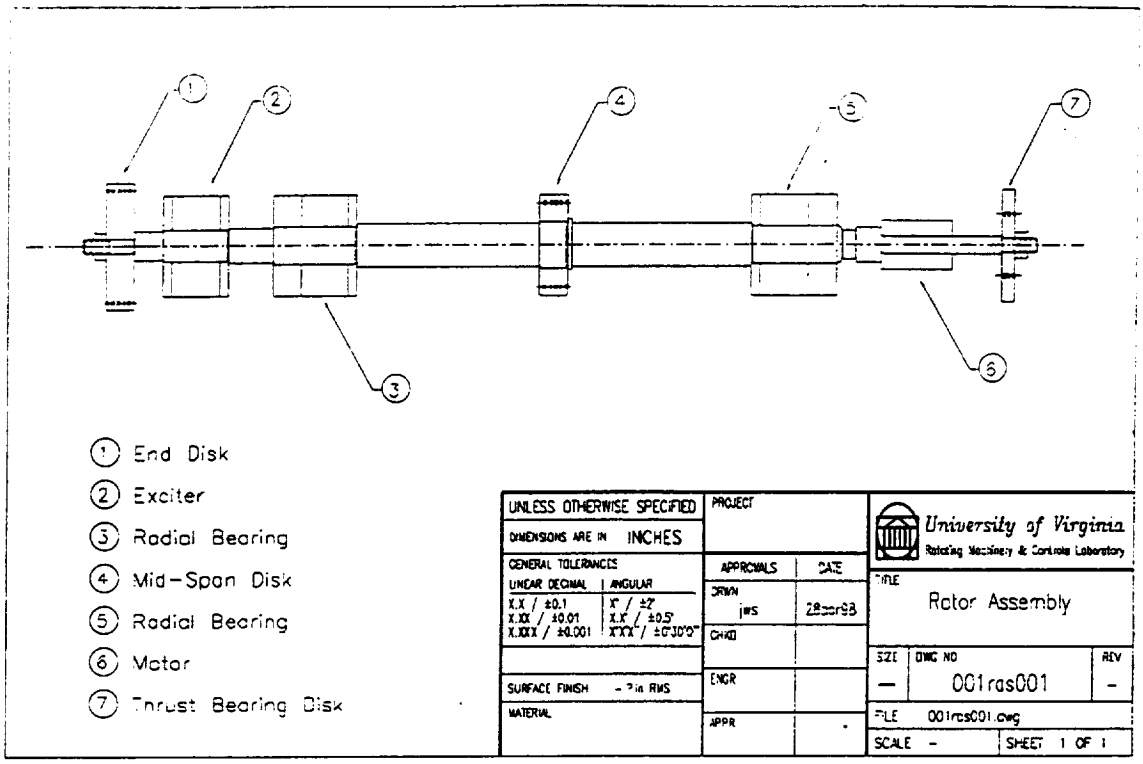

Figure 5: Detail of the rotor for the UVA/ROMAC controls test rig

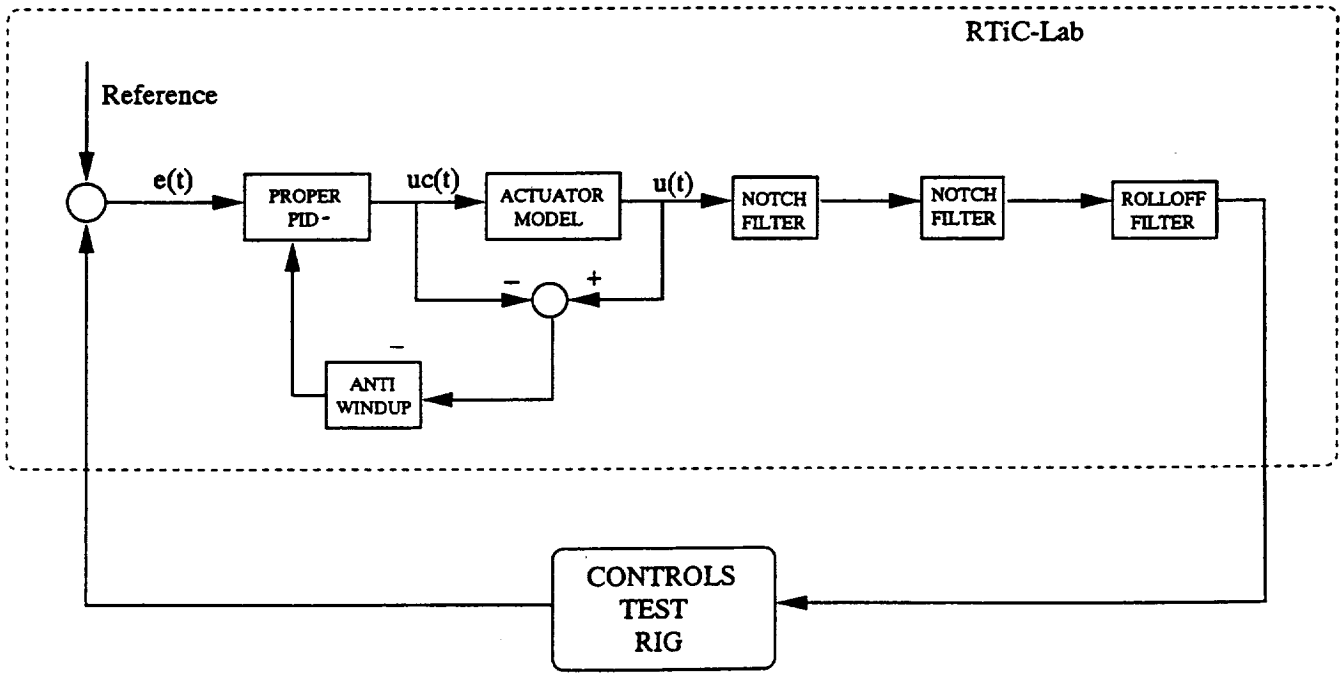

Figure 6: Detail of each of the axis controllers

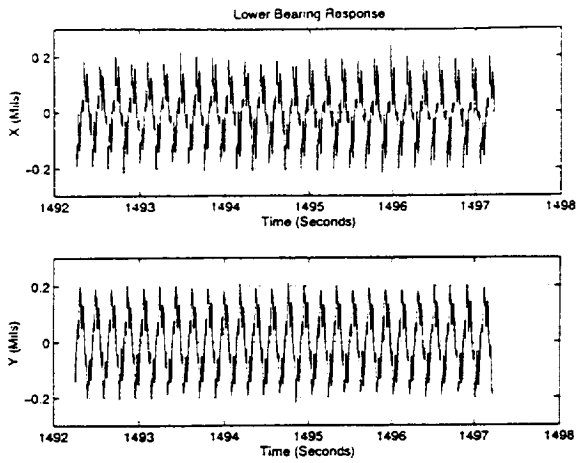

Figure 7: Lower bearing response at 5,000 RPM

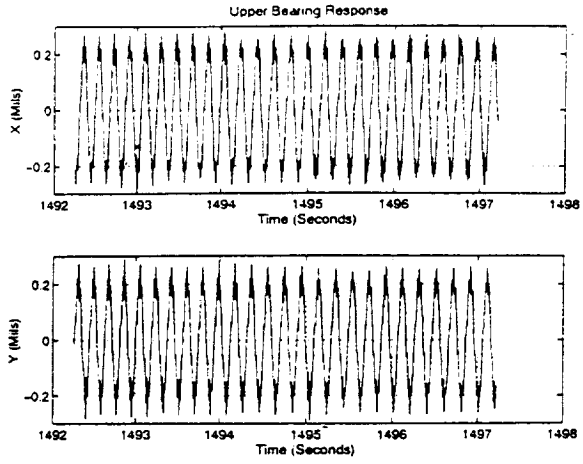

Figure 8: Upper bearing response at 5,000 RPM

Figure 9 shows the actual predictability of RT-Linux. Here, it is shown that the sampling error at 5 kHz sampling is small and bounded to less than a 0.6%. From (1), the CPU utilization of our DCC is 62% and well below its 69% bound.[7]

4 Conclusion

Basic understanding of Real Time Systems scheduling results is necessary towards efficient and successful implementation of *predictable* real time controllers for AMB systems. Namely, for periodic tasks, such as fixed rate controllers, it has been shown that priority assignment for a fixed priority scheduling algorithm via the Liu and Layland RMA will lead to predictable control systems for AMBs. Most importantly, via this priority assignment policy, it is possible to implement multiple tasks that are used not only for the actual suspension control but also for real time monitoring, data logging, data display, network communications, and controller parameter updates.

In order to simplify controller implementation by use of these scheduling techniques, the Real Time Controls Laboratory, or RTiC-Lab, was developed at UVA/ROMAC which aids in the controller implementation process. It uses both Linux and Real Time Linux as the implementation platform. And, consistent with the underlying operating system, RTiC-Lab is Open Source, which means that AMB controller developers who are interested in using this software are encouraged to download the software from `http://www.people.virginia.edu/~efh4v`, use it, and contribute to it by sending modifications to `efhilton@alum.mit.edu`.

References

P. E. Allaire, H. C. Kim, E. H. Maslen, G. B. Bearnson, and D. B. Olsen. Design of a magnetic bearing supported prototype centrifugal artificial heart pump. In *Tribology Transactions*, volume 39-3, pages

[7]Note that for this particular controller system, this level of utilization shows that our computer was properly sized for this controller algorithm although may be undersized for a more complex or higher order controller algorithm.

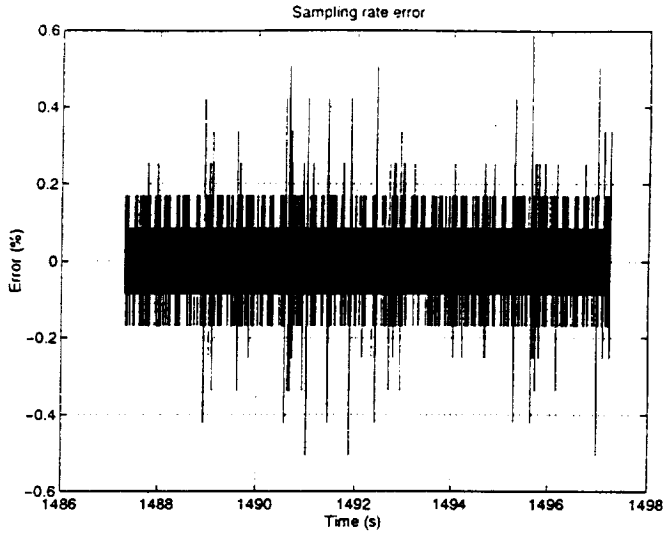

Figure 9: Sampling error of RTiC-Lab during rotor suspension at 5,000 RPM

663–669, July 1996.

P. E. Allaire, E. H. Maslen, R. R. Humphris, C. R. Knospe, and D. W. Lewis. Magnetic bearings. In E. R. Boozer, editor, *CRC Handbook of Lubrication and Tribology*, volume 3, pages 577–600, 1994.

M. Baloh, P. Allaire, E. Hilton, N. Wei, E. Maslen, D. Baun, and R. Flack. Magnetic bearing system for continous flow ventricular assist device. *J. of the ASAIO*, 45(5), 1999.

R. Bartlett, J. Coyner, S. Djouadi, P. Allaire, E. Hilton, J. Luo, P. Tsiotras, F. Maher, and R. Strunce. A simulation of spacecraft energy momentum wheels using advanced **magnetic** bearing controllers. In *1999 Invitational NASA/Air Force Flywheel Workshop*, NASA Glenn Research Center, 1999. Accepted for Publication, Invited Paper.

E.G. Coffman and P.J Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

R.W Conway, W.L. Maxwell, and Miller L.W. *Theory of Scheduling.* Addison-Wesley, 1967.

E. Hilton, P. Allaire, M. Baloh, N. Wei, G. Bearnson, D. Olsen, and P. Khanwilkar. Test controller design, implementation, and performance for a magnetic suspension continous flow ventricular assist device. *Artificial Organs*, 23(8):785–791, 1999.

J.P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behaviour. *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

J. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10), October 1988.

J. Stankovic and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6), June 1995.

J. Stankovic and K. Ramamritham. What is predictability for real-time systems? *J. Real-Time Systems*, 2:247–254, December 1990.